BinOpが二項演算を表すノード
子ノードはleft,op,rightとなっており、opには演算子が入り、opの左側の項がleft、右側の項がrightに入るようになっている
leftとrightには数値(変数または整数または実数)かBinOpが入る
BinOpは、最も深い場所にある葉の二項演算から実行されるように格納される（根が一番最後に実行される）
pythonの/演算子はrubyでは数式/数式.to_fと等価であり、//演算子は(数式/数式).to_iとほぼ等価である．(ただし、数値∈数式である)

構文規則(型で定義されている部分はその型の範囲全て)

$\odot$:== $+ \mid - \mid * \mid **$

e :== $d \mid f \mid e \odot e \mid e / e \mid e // e \mid (e)$

d :== $[\![int]\!]^{Python}$

f :== $[\![float]\!]^{Python}$

$[\![int]\!]^{Python}$ の制限はメモリサイズ依存

$[\![float]\!]^{Python}$ は $[\![double]\!]^{C}$ で実装されている

数値演算翻訳
IはLatexのMathcalに直す．字句は普通の書式の太字に直す

$I\ [\![e_1 \odot e_2]\!]^{Python} = (I\ [\![e_1]\!]^{Python} I\ [\![\odot]\!]^{Python} I\ [\![e_2]\!]^{Python})$

$I\ [\![(e)]\!]^{Python} = (I\ [\![e]\!]^{Python})$

$I\ [\![e_1/e_2]\!]^{Python} = (I\ [\![e_1]\!]^{Python} I\ [\![/]\!]^{Python} I\ [\![e_2]\!]^{Python}.to\_f)$

$I\ [\![e_1//e_2]\!]^{Python} = (I\ [\![e_1]\!]^{Python} I\ [\![//]\!]^{Python} I\ [\![e_2]\!]^{Python}).to\_i$

$I\ [\![+]\!]^{Python} = [\![+]\!]^{Ruby}$

$I\ [\![-]\!]^{Python} = [\![-]\!]^{Ruby}$

$I\ [\![*]\!]^{Python} = [\![*]\!]^{Ruby}$

$I\ [\![/]\!]^{Python} = [\![/]\!]^{Ruby}$

$I\ [\![//]\!]^{Python} = [\![/]\!]^{Ruby}$

$I\ [\![**]\!]^{Python} = [\![**]\!]^{Ruby}$

$I\ [\![int]\!]^{Python} = [\![Integer]\!]^{Ruby}$

$I\ [\![float]\!]^{Python} = [\![Float]\!]^{Ruby}$

そのため、pythonのBinOpをrubyの四則演算に翻訳するには
1.'('を出力する
2.leftの中身をチェックし、数値ならその数値を出力、BinOpなら再帰する
3.opの中身をチェックし、/と//以外ならそのまま出力、/か//だった場合は種類に応じたフラグをセットし、'/'を出力する
4.rightの中身をチェックし、数値ならその数値を出力、BinOpなら再帰する
5./のフラグがセットされていた場合、'.to_f'を出力する
6.')'を出力する
7.//のフラグがセットされていた場合、'.to_i'を出力する
という働きをする関数を作成すればよい
なお、この時点では関数の存在について言及していないため、後に関数入りの四則演算にも

対応させる拡張が必要

出力した四則演算が最も深い葉の二項演算から実行される証明
leftの深さをn,rightの深さを1とする。leftノードが数値であったとき、任意の数値l1,l2,…,lnが入る。rightノードも同様にr1,r2,…,rnが入る。opは任意の演算子が入るが、ここでは簡単化のため+とする。
1)n=1のとき
BinOp
｜ ｜
l1 r1
となり、実行結果は
(l1+r1)
となるため、正しい

2)n=kのとき、与えられた条件が正しいと仮定する
n=k+1のとき
BinOp
｜ ｜
BinOp r1
｜ ｜
BinOp r2
｜ ｜
…
｜ ｜
l1 rk+1
となる
関数が最も深いノードの処理を終えたときの出力結果は
(((…(l1+rk+1)
↑k+1個
となり、最も深い葉の二項演算が最初に実行される
更に、最も深い葉の処理が終了しているので、残りの深さはnとなる。仮定より、深さnのときにこの関数の出力は正しいので、残りの部分の出力も正しい。
1),2)より、この関数はleftノードの深さがn,rightノードの深さが1のとき、抽象構文木から四則演算を正しく具象化して出力することができる

rightノードについても同様の証明ができるため、この関数は正しく動く

字句解析と字句の変更のみで翻訳を行う場合、四則演算は構造が分からないと翻訳できないため、一度逆ポーランド記法などの構造が分かるものに変換する必要がある。
仮に逆ポーランド記法に直したとすると、式を左から読んでいき、数値が来たらスタックにその数値をpush、演算子が来たらスタックから二回popを行い、'()'を付けた二項演算にして、その二項演算をスタックにpush。このとき、演算子が/か//だった場合は上記にあるような式になるような処理を行う。式がなくなった時点でスタックに唯一残るものが求める式である。

Ifがif文を表すノード
target,body,orelseの子ノードを持つ
targetは論理式が入る
bodyはtargetがTrueだったときに実行される処理が入る
orelseはtargetがFalseだったときに実行される処理が入る
if文にelif句があったとき、elif句からelse句まではIfノードとしてorelseノードに格納される
例：

```
if i==0:
    pass
elif i==1:
    pass
elif i==2:
    pass
else:
    pass
```

rubyに翻訳するためには
0) 再帰が何回行われたか数える変数を引数として用意しておく
1) orelseノードの長さが1でない、またはIfでないかチェックし、変数に保存
2) 再帰が0回のとき、'if 'を出力。そうでないとき、'elif 'を出力
3) 論理式をチェックし、出力する関数を、targetを引数として呼び出す
4) ' then\n'を出力
5) bodyの中身を一つずつチェックし、そのノードの種類に応じた処理を行う関数を、引数をbodyとして呼び出す
6) 1)の変数がFalseだったときorelse[0],0)の変数+1を引数として再帰。そうでないとき、'else\n'を出力し、5)の関数の引数をorelseとして呼び出す
7) 1)の変数がTrueだった場合'end'を出力。そうでなかったとき、何もしない。
8) '\n'を出力
という関数を作ればよい。

正しく翻訳されることの証明
一つのif～elif～else文に対し、一つのendが出力されることを証明する。ただし、5)の関数は正しく動くと仮定し、1)の変数名はif_checkとする。
再帰の回数をnとすると
1) n=0のとき
n=0なので'if 論理式 then\n'が出力される。その後、bodyの処理が出力される。
if_checkがFalseとなるので、'else\n'が出力される。その後、orelseの処理が出力される。
if_checkがFalseなので、最後に'end'が出力され、最終的な出力結果は

```
if 論理式 then
bodyの中身
else
orelseの中身
end
```

となり、これは正しい

2) n=1のとき
n=1なので一回再帰する。出力は上記のbodyの中身までは同じだが、n=1なので
if_check=Trueとなり、再帰する。この時点でn=1となるため、

if_stmt :== 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
suiteは基本的に文であるため，まだ考えない
testは論理式であるため，まだ考えない